Feeding of the Thousands
–
Leveraging the GPU's Computing Power for Sparse Linear Algebra

Hartwig Anzt
Sparse Linear Algebra on GPUs

- Inherently parallel operations
  - axpy, copy, gemv...
  - usually memory bound
  - Kernel fusion

- Sparse matrix vector product
  - often computationally most expensive part
  - variety of storage formats, kernels...
  - component-thread mapping can result in imbalance
  - malicious memory access

- Bottlenecks
  - sequential operations, unstructured, random memory access
  - incomplete factorizations (ILU/IC)
  - sparse triangular solves
Sparse Matrix Vector Product (SpMV)

- Sliced Ellpack (SELL) format as trade-off between CSR and Ellpack
- Sorting can improve load-balancing

Sparse Matrix Vector Product (SpMV)

- Assign multiple threads to each row
- 2-dimensional thread blocks

Sparse Matrix Vector Product with multiple Vectors (SpMM)

- 3-dimensional thread blocks for processing multiple vectors simultaneously

Sparse Matrix Vector Product with multiple Vectors (SpMM)

- 3-dimensional thread blocks for processing multiple vectors simultaneously
- Performance on NVIDIA K40, 64 vectors, DP:

Kernel Fusion in Sparse Iterative Algorithms

- Memory bandwidth in many cases the performance bottleneck.
- Sequence of consecutive vector updates (BLAS 1) benefits from enhanced data locality.
- Design of algorithm-specific kernels.
Kernel Fusion in Sparse Iterative Algorithms

- Memory bandwidth in many cases the performance bottleneck
- Sequence of consecutive vector updates (BLAS 1) benefits from enhanced data locality

```c
while( ( k < maxiter ) && ( res > epsilon ) ){
    Scalar_SpMV <<<Gs, Bs>>> ( n, rowA, colA, valA, d, z );
    tmp = cublasSdot ( n, d, l, z, 1 );
    rho = beta / tmp;
    gamma = beta;
    cublasSaxpy ( n, rho, d, l, x, 1 );
    cublasSaxpy ( n, -rho, z, l, r, 1 );
    tmp = cublasSdot ( n, r, l, r, 1 );
    beta = tmp;
    alpha = beta / gamma;
    cublasSscal ( n, alpha, d, 1 );
    cublasSaxpy ( n, one, r, l, d, 1 );
    res = sqrt( beta );
    k++;
} // end-while

while( ( k < maxiter ) && ( res > epsilon ) ){
    scalar_fusion_1 <<<Gs, Bs, Ms>>> ( n, rowA, colA, valA, 
                                      d, z, beta, rho, gamma, vtmp );
    fusion_2 (Gs, Bs, Ms, n, beta, rho, vtmp );
    fusion_3 <<<Gs, Bs, Ms>>> ( n, rho, d, x, z, r, vtmp );
    fusion_4 (Gs, Bs, Ms, n, vtmp, vtmp2 );
    fusion_5 <<<Gs, Bs>>> ( n, beta, gamma, alpha, 
                          d, r, vtmp );
    cudaMemcpy( &res, beta, sizeof(float), cudaMemcpyDeviceToHost );
    cudaMemcpyDeviceToHost( res, sqrt( beta );
    k ++;
} // end-while
```

Aliaga et al.: Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs, Parallel Processing (ICPP), 2013.
Kernel Fusion in Sparse Iterative Algorithms

- Which operations can be merged into a single kernel?
  - kernels compatible in terms of component-thread mapping
  - example classification for Jacobi-CG:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input vector(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXPY</td>
<td>$y = \alpha x + y$</td>
<td>mapped, mapped, mapped</td>
</tr>
<tr>
<td>COPY</td>
<td>$y = x$</td>
<td>mapped, mapped, mapped</td>
</tr>
<tr>
<td>DOT</td>
<td>$\alpha = \langle x, y \rangle$</td>
<td>mapped, mapped, unmapped</td>
</tr>
<tr>
<td>Jacobi-Prec</td>
<td>$y = M^{-1}x$</td>
<td>mapped, mapped, mapped</td>
</tr>
<tr>
<td>SpMV CSR</td>
<td>$y = Ax$</td>
<td>unmapped, -     mapped</td>
</tr>
<tr>
<td>SpMV ELL</td>
<td>$y = Ax$</td>
<td>unmapped, -     mapped</td>
</tr>
<tr>
<td>SpMV SELL-P</td>
<td>$y = Ax$</td>
<td>unmapped, -     unmapped</td>
</tr>
</tbody>
</table>

Kernel Fusion in Sparse Iterative Algorithms

- Which operations can be merged into a single kernel?
- Which kernels do we want to merge?
  - performance vs. flexibility...

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input vector(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXPY</td>
<td>( y = \alpha x + y )</td>
<td>mapped</td>
</tr>
<tr>
<td>COPY</td>
<td>( y = x )</td>
<td>mapped</td>
</tr>
<tr>
<td>DOT</td>
<td>( \alpha = \langle x, y \rangle )</td>
<td>mapped</td>
</tr>
<tr>
<td>Jacobi-Prec</td>
<td>( y = M^{-1}x )</td>
<td>mapped</td>
</tr>
<tr>
<td>SpMV CSR</td>
<td>( y = Ax )</td>
<td>unmapped</td>
</tr>
<tr>
<td>SpMV ELL</td>
<td>( y = Ax )</td>
<td>unmapped</td>
</tr>
<tr>
<td>SpMV SELL-P</td>
<td>( y = Ax )</td>
<td>unmapped</td>
</tr>
</tbody>
</table>

Code Jacobi-preconditioned J-CG, J-BiCGSTAB, J-IDR, J-GMRES...? How about ILU/IC?
Kernel Fusion in Sparse Iterative Algorithms

- Which operations can be merged into a single kernel?
- Which kernels do we want to merge?
  - performance vs. flexibility...

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input vector(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXPY</td>
<td>( y = \alpha x + y )</td>
<td>mapped</td>
</tr>
<tr>
<td>COPY</td>
<td>( y = x )</td>
<td>mapped</td>
</tr>
<tr>
<td>DOT</td>
<td>( \alpha = \langle x, y \rangle )</td>
<td>mapped</td>
</tr>
<tr>
<td>Jacobi-Prec</td>
<td>( y = M^{-1}x )</td>
<td>mapped</td>
</tr>
<tr>
<td>SpMV CSR</td>
<td>( y = Ax )</td>
<td>unmapped</td>
</tr>
<tr>
<td>SpMV ELL</td>
<td>( y = Ax )</td>
<td>unmapped</td>
</tr>
<tr>
<td>SpMV SELL-P</td>
<td>( y = Ax )</td>
<td>unmapped</td>
</tr>
</tbody>
</table>

One stand-alone code for each SpMV kernel?
Kernel Fusion in Sparse Iterative Algorithms

- Which operations can be merged into a single kernel?
- Which kernels do we want to merge?
  - performance vs. flexibility...

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Size</th>
<th>Nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUD</td>
<td>943,695</td>
<td>77,651,847</td>
</tr>
<tr>
<td>G3</td>
<td>1,585,478</td>
<td>7,660,826</td>
</tr>
<tr>
<td>INL</td>
<td>503,712</td>
<td>36,816,342</td>
</tr>
<tr>
<td>LDO</td>
<td>952,203</td>
<td>46,522,475</td>
</tr>
</tbody>
</table>

- Benefits from fusion Jacobi-preconditioner for very large and sparse matrices.
- Smaller benefits for more complex algorithms (BiCGSTAB, CGS, QMR, IDR...)

![Runtime Chart]

**Legend:**
- basic JCG
- fusion JCG
- Jacobi-fusion
Kernel Fusion in Sparse Iterative Algorithms

- How close can kernel fusion bring us to the **theoretical performance bound induced by memory bandwidth**?
- Cooperation with *Moritz Kreutzer, Eduardo Ponce*.
- NVIDIA K40, theoretical bandwidth: 288 GB/s...

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR</td>
<td>170,998</td>
</tr>
<tr>
<td>TDK</td>
<td>204,316</td>
</tr>
<tr>
<td>WEB</td>
<td>1,000,005</td>
</tr>
<tr>
<td>DIE</td>
<td>1,157,456</td>
</tr>
</tbody>
</table>
Kernel Fusion in Sparse Iterative Algorithms

- **Efficiency compared to roofline performance model:** \( P = \min(P^{\text{peak}}; Ib) \) Gflop/s

\( P \) theoretical compute peak, \( I \) intensity, \( b \) bandwidth

**IDR(s) general Krylov solver**

*Moritz Kreutzer, Eduardo Ponce*

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR</td>
<td>170,998, 958,936</td>
</tr>
<tr>
<td>TDK</td>
<td>204,316, 2,846,228</td>
</tr>
<tr>
<td>WEB</td>
<td>1,000,005, 3,105,536</td>
</tr>
<tr>
<td>NLP</td>
<td>1,062,400, 28,704,672</td>
</tr>
<tr>
<td>DIE</td>
<td>1,157,456, 48,538,952</td>
</tr>
<tr>
<td>THM</td>
<td>1,228,045, 8,580,313</td>
</tr>
<tr>
<td>AFS</td>
<td>1,508,065, 52,672,325</td>
</tr>
<tr>
<td>MLG</td>
<td>1,504,002, 110,879,972</td>
</tr>
<tr>
<td>G3</td>
<td>1,585,478, 7,660,826</td>
</tr>
<tr>
<td>TRA</td>
<td>1,602,111, 23,500,731</td>
</tr>
</tbody>
</table>
Kernel Overlap in Sparse Iterative Algorithms

- **Concurrent kernel execution** to exploit unused GPU compute resources.
- **Rare in sparse linear algebra** (most algorithms compose of memory-bound operations).
- **Small benefits** if datasets too small to saturate memory bandwidth.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR</td>
<td>170,998</td>
</tr>
<tr>
<td>TDK</td>
<td>204,316</td>
</tr>
<tr>
<td>WEB</td>
<td>1,000,005</td>
</tr>
<tr>
<td>DIE</td>
<td>1,157,456</td>
</tr>
</tbody>
</table>

![Bar graph showing performance improvement for different matrices and techniques.](image)
Bottlenecks

- **Operations not parallelizable to GPU thread concurrency:**
  - sequential operations
  - unstructured, random memory access
  - incomplete factorizations (ILU/IC)
  - sparse triangular solves
Bottlenecks

- **Operations not parallelizable to GPU thread concurrency:**
  - sequential operations
  - unstructured, random memory access
  - incomplete factorizations (ILU/IC)
  - sparse triangular solves

- **Don’t even try – rethink the problem!**
  - A different algorithm may take you to the same goal.
  - Choose algorithms with fine-grained parallelism, avoid synchronizations.
  - Sparse iterative solvers provide approximations -- relax the bit-wise reproducibility criterion.

Most popular Example: *Iterative ILU* (Chow et al.)

Bottlenecks

- Example: sparse triangular solves in ILU preconditioning:
  - inherently sequential
  - parallelism from level-scheduling/multi-color ordering
  - unable to exploit fine-grained parallelism of GPUs
Bottlenecks

- **Example:** sparse triangular solves in ILU preconditioning:
  - inherently sequential
  - parallelism from level-scheduling/multi-color ordering
  - unable to exploit fine-grained parallelism of GPUs

- **Take an unconventional approach:**
  Approximate sparse triangular solves

- Replace forward/backward substitutions with **iterative method.**

- **Low solution accuracy required** as $\text{LU} \approx A$ typically only a rough approximation.

- Better **scalability** of iterative methods.

- **Jacobi converges** as spectral radius of iteration matrix smaller 1:

  $$x^{k+1} = D^{-1}b + Mx^k$$

  $$M_L = D_L^{-1}(D_L - L) = I - L$$

  $$M_U = D_U^{-1}(D_U - U) = I - D_U^{-1}U$$

- **Performance depends on SpMV.**
Bottlenecks

- Example: sparse triangular solves in ILU preconditioning:
  - inherently sequential
  - parallelism from level-scheduling/multi-color ordering
  - unable to exploit fine-grained parallelism of GPUs

- Take an unconventional approach: Approximate sparse triangular solves

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Exact IC</th>
<th>10 Jacobi sweeps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level PCG:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Laplace 3D 27pt</td>
<td>58</td>
<td>1.83</td>
</tr>
<tr>
<td>parabolic_fem</td>
<td>645</td>
<td>37.24</td>
</tr>
<tr>
<td>thermal2</td>
<td>1771</td>
<td>305.58</td>
</tr>
<tr>
<td>G3_circuit</td>
<td>1625</td>
<td>45.60</td>
</tr>
</tbody>
</table>

Anzt et al., *Iterative Sparse Triangular Solves for Preconditioning*, Euro-Par 2015.
Summary

- **SpMV optimization** central challenge in iterative sparse linear algebra.
- Algorithms **memory bound**, often benefit from **kernel fusion**.
- Trade-off between **performance** and **flexibility**:
  - SpMV /Jacobi as building block enhances modularity.
- **Kernel fusion** can bring performance close to **theoretical bound**.
- Concurrent kernel execution only beneficial for small problems.
- We need **unconventional approaches** for **bottleneck-operations**.
  - **Iterative ILU** generation (Chow et al.)
  - **Iterative sparse triangular solves** for ILU/IC.

This research is based on a cooperation with Enrique Quintana-Ortí from the University Jaume I, Edmond Chow from Georgia Tech, and Moritz Kreutzer from the University of Erlangen.