Batched Sparse Iterative Solvers for Computational Chemistry Simulations on GPUs

ScalA 21
November 19, 2021
Batched Sparse Iterative Solvers for Computational Chemistry Simulations on GPUs

Isha Aggarwal, Aditya Kashi, Pratik Nayak, Cody J. Balos, Carol S. Woodward and Hartwig Anzt

November 19th, 2021

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.
Direct methods vs. iterative methods

Direct methods
- “Pre-defined instruction execution order”
- No initial guess needed
- Methods can be implemented as self-contained, especially for dense

Iterative methods
- No pre-defined instruction execution order; algorithm execution dependent on input (convergence depends on eigenvalues, etc.)
  - Need to monitor convergence
- Initial guess can reduce iteration count
  - Need to allow for providing initial guess
- Methods use additional components that each have different options (preconditioners, SpMV kernel…)
  - Needs to allow for interfacing the required functionality
Direct methods vs. iterative methods

**Direct methods**
- “Pre-defined instruction execution order”
- No initial guess needed
- Methods can be implemented as self-contained, especially for dense

**Iterative methods**
- No pre-defined instruction execution order; algorithm execution dependent on input (convergence depends on eigenvalues, etc.)
  - Need to monitor convergence
- Initial guess can reduce iteration count
  - Need to allow for providing initial guess
- Methods use additional components that each have different options (preconditioners, SpMV kernel...)
  - Needs to allow for interfacing the required functionality

**C++ direct solver interface**
```
solution = solver(matrix, rhs);
```

*single-kernel implementation is “straight-forward”*

**C++ iterative solver interface**
```
solution = solver(matrix, rhs,
                  initial_guess, SpMV_kernel,
                  preconditioner, stopping_criterion);
```

*much flexibility and external component interfacing needed*

*Data exchange usually realized via main memory*
Batched Iterative Solver Setting

- Many sparse problems of medium size have to be solved concurrently.
  - ~ 10 – 1000 unknowns (rows)
  - All sparse systems may share the same sparsity pattern
  - An approximate solution may be acceptable (e.g., inside a non-linear solver)

- One solution is to arrange the individual systems into one large block-diagonal system.
  - Convergence determined by the “hardest” problem
  - No reuse of sparsity pattern information
  - Global synchronization points
  - May need pre-processing of pointer and index arrays

- Better approach: design batched iterative solve functionality that solves problems asynchronously.
  - Problem-dependent convergence accounted for
  - No global synchronization
  - Reuse of sparsity pattern information
  - Parallelize across individual problems
## Performance aspects of batched kernels

1. **Batched functionality is generally memory-bound**  
   -> *urgent need to minimize main memory access*

2. **Different problems and solvers have different resource requirements**  
   -> *need to predict the shared memory requirement*

3. **Different problems may result in different algorithm behavior**

### Implication for sparse iterative methods

- Interfacing solver components via main memory impacts performance
- Single, monolithic kernel necessary for communication via shared memory; not straightforward for iterative solvers

- Sparse matrix memory needs unknown
- Shared memory requirement is variable across both problem sizes and solver types
- Caching can only be used for constant data
- Register usage frequently becomes a bottleneck.

- Need to monitor iterative solver convergence for each problem individually and exit when done
Batched iterative solver design in Ginkgo

Design Choices

- Single solver kernel are templated (C++) on matrix type, preconditioner type, stopping criterion and logger.
- Solver iteration loop, including all components, in device code.
- Only core solver classes are part of user interface.
- User chooses preconditioner type etc. by enum-type factory parameters to the solver classes; used to dispatch to correct kernel instantiation.
- Modular architecture, but not possible for users to supply their own preconditioners. New preconditioners etc. must be added in Ginkgo.
- Multiple right hand sides are not supported.
Batched kernel execution in Ginkgo

Mapping to execution units and memory levels:

- Each thread block handles the solution of one system.
- No synchronization between individual solver completions.
- Shared memory is used for intermediate vectors (r/w).
- Const data (matrix, RHS) is in global memory (+ L1 cache).
- Logger called only once to log number of iterations and final implicit residual norm for every system in the batch.
- More detailed loggers possible but come with lower performance.
First experiences with Ginkgo’s batched iterative solvers in PeleLM

PeleLM is a parallel, adaptive mesh refinement (AMR) code that solves the reacting Navier-Stokes equations in the low Mach number regime.

AMReX source code
https://amrex-combustion.github.io/PeleLM/overview.html

Ginkgo solvers benchmarked on NVIDIA V100 on Summit.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Size</th>
<th>Non-zeros (A)</th>
<th>Non-zeros (L+U)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dodecane_Lu</td>
<td>54</td>
<td>2,332 (80%)</td>
<td>2,754 (94%)</td>
</tr>
<tr>
<td>drm19</td>
<td>22</td>
<td>438 (90%)</td>
<td>442 (91%)</td>
</tr>
<tr>
<td>gri12</td>
<td>33</td>
<td>978 (90%)</td>
<td>1,018 (93%)</td>
</tr>
<tr>
<td>gri30</td>
<td>54</td>
<td>2,560 (88%)</td>
<td>2,860 (98%)</td>
</tr>
<tr>
<td>isoctane</td>
<td>144</td>
<td>6,135 (30%)</td>
<td>20,307 (98%)</td>
</tr>
<tr>
<td>lidryer</td>
<td>10</td>
<td>91 (91%)</td>
<td>91 (91%)</td>
</tr>
</tbody>
</table>
Iteration counts with batched iterative solvers on PeleLM matrices

(a) dodecane_lu

(b) drm19

(c) gri12

(d) gri30

(e) isoctane

(f) lidryer
Relative residuals with batched iterative solvers on PeleLM matrices

(a) dodecane.lu
(b) drm19
(c) gri12
(d) gri30
(e) isoctane
(f) lidryer
Timing comparisons

(a) dodecane_lu
(b) drm19
(c) gri12
(d) gri30
(e) isoctane
(f) lidryer
First experiences with Ginkgo’s batched iterative solvers in PeleLM

(a) dodec
(b) gri12
(d) gr
(e) nododecanelu
(f) lidryer
First experiences with Ginkgo’s batched iterative solvers in PeleLM

Speedups in production code may be larger:
- Use of “suitable” initial guess (last linear solve)
- Less accurate solution often sufficient
Conclusions

- Batched sparse iterative solvers are effective for computational chemistry problems in Pele-LM.

- Ginkgo implementation gives good speedups over batched dense LU for both very sparse problems (eg. isooctane) as well as denser problems (eg. gri30) on NVidia V100.

- Achieved along with some of the flexibility expected of iterative solvers:
  - Choice of matrix formats
  - Choice of preconditioner
  - Choice of two convergence criteria
Limitations and upcoming work

- Application is assumed to pass data in contiguous memory.

- Size of each linear system was limited by size of shared memory.
  - New: Split intermediate vectors in the solver among shared and global memory.

- Currently, only Jacobi preconditioner. More in the pipeline.

- New: HIP implementation for AMD GPUs is available.

- Individual system scheduling handled by GPU runtime.
  - How can we tell the runtime to schedule harder problems first?
  - How do we identify the harder problems?

- Is there a need to support multiple RHS?
  - One might prefer a direct solver