Symmetric Tridiagonal Eigenvalue Problem on Multi-GPU Systems

Hyunsu Cho and Peter Yoon
Department of Computer Science, Trinity College, Hartford, CT

Problem

Find the eigenvalues and eigenvectors of a symmetric tridiagonal matrix *T*. That is, find its **eigendecomposition**

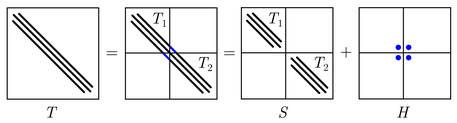
$$T = QDQ^T$$

where *Q* is orthogonal and *D* is diagonal. *D* has the eigenvalues in the diagonal and *Q* has the eigenvectors in its columns.

Divide-and-Conquer Algorithm

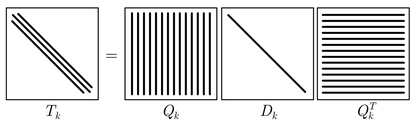
1. Divide

Divide the problem until we reach **base cases**: *k* x *k* tridiagonal systems where *k* is small.



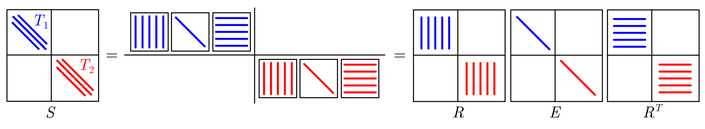
2. Conquer

Decompose the base cases using QR.

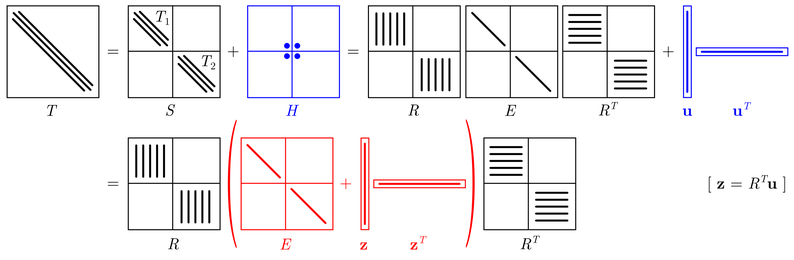


3. Merge

Build a partial solution *S* from two eigendecompositions.



Perform **rank-one update** on *S* to take account of *H*.



The inner system (red) is then decomposed by the following stages. See [1] for more details.

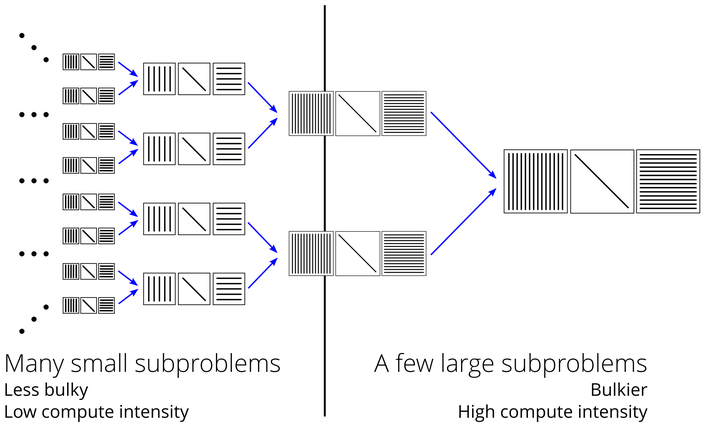
1. Deflate the eigenvalues and eigenvectors that don't need to be explicitly computed. *Inherently serial (permutation)*
2. Solve the **secular equation** to compute the eigenvalues. *Parallelizable*
3. Solve an inverse eigenvalue problem to recover the eigenvectors of the inner system. *Parallelizable*
4. Recover the eigenvectors of *T* by computing *Q* = *RU*, where *U* has the eigenvectors collected in Stage 3. *Highly parallelizable (BLAS 3)*
5. Reorder the deflated eigenvalues/eigenvectors into their place. *Inherently serial (permutation)*

Divide-and-conquer algorithm is a numerically stable and efficient algorithm that computes the eigenvalues and eigenvectors of a **symmetric tridiagonal matrix**. A major challenge in implementing the algorithm on multiple GPUs is the **low compute intensity** at the bottom of the divide-and-conquer tree, where the subproblem size is small. Conventional implementations on multi-GPU systems fall short of addressing this issue, leaving GPUs idle much of the time. We overcome the problem by merging multiple pairs of subproblems in parallel. Preliminary runs show promising results. Our implementation running on 4 GPUs shows a 12x speedup over the sequential counterpart. Furthermore, it exhibits a meaningful degree of scaling with respect to the number of GPUs, running 2x as fast on 4 GPUs as on 1 GPU.

Computational Challenge

Compute intensity is low when the algorithm has just started merging up from the base cases.

- The algorithm needs to tackle many small subproblems first before proceeding to larger ones.
- Small subproblems are **not sufficiently bulky** to keep multiple GPUs busy at the same time.
- By the time the algorithm reaches bulky subproblems, it has only a few merge operations to do – the number of subproblems halves at each level.



- Conventional multi-GPU implementations, such as one in the MAGMA library [2, 3], split each merge task evenly among multiple GPUs.
- A single merge task is often too small and does not give GPUs enough work to do. As a result, performance scales poorly with respect to the number of GPUs used.

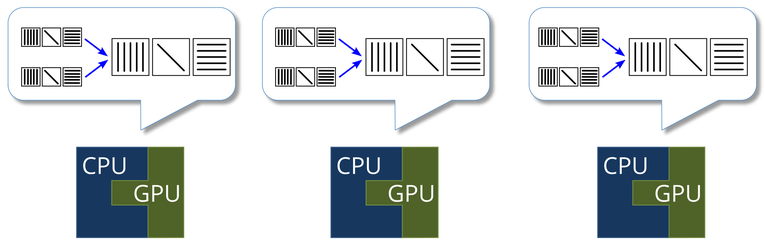
References

[1] J. Demmel. *Applied Numerical Linear Algebra*, page 216-226
[2] MAGMA (Matrix Algebra on GPU and Multicore Architectures) Library, version 1.4.1. URL: http://icl.cs.utk.edu/magma/
[3] C. Vomel, S. Tomov, J. Dongarra. "Divide and Conquer on Hybrid GPU-Accelerated Multicore Systems," *SIAM Journal on Scientific Computing*, 34 (2), C70-82, April 12, 2012.

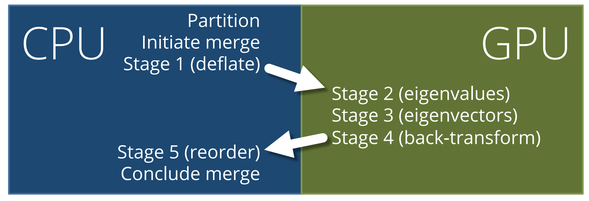
Our Approach

Use multiple GPUs to merge **multiple pairs** of subproblems in parallel.

- Form **compute groups**. Each compute group consists of one GPU device and one CPU thread.
- Each compute group performs one merge task.

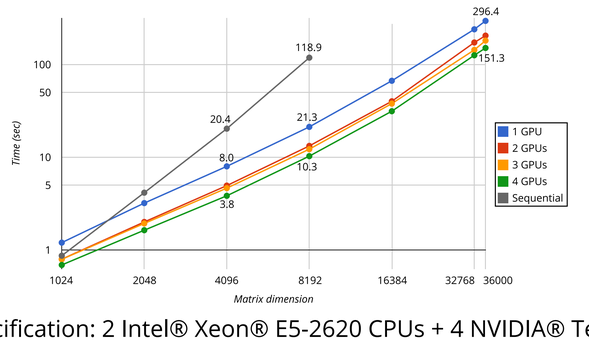


- Compute intensity is improved as GPUs no longer have to compete over limited amount of work; instead, we multiply the workload to match the number of GPUs.



- When merge sizes grow large, we may want to switch to the conventional scheme where compute groups tackle one merge task at a time.
- If subproblems grow too large to fit into GPU memory (e.g. 16384), we have to peform the tasks out-of-core. For now, almost all tasks run on CPUs, except for matrix multiplication in Stage 4, which is done by GPUs.

Preliminary Results



- Specification: 2 Intel® Xeon® E5-2620 CPUs + 4 NVIDIA® Tesla® K20c GPUs + 64 GB main memory + 5GB per-GPU memory
- 16x speedup over sequential version when run on 4 GPUs
- Meaningful degree of scaling with respect to number of GPUs: 4 GPUs twice as fast as 1 GPU
- Memory transfer between host and device accounts for 14.8% of compute time.

Future work

- Optimize the GPU kernels for secular equation solver (Stage 2).
- Divide parallel tasks (Stages 2-4) among CPUs as well as GPUs.
- Form additional compute groups with leftover CPU cores.