Title
A GPU-Accelerated Structurally-Symmetric Sparse Multifrontal Solver

Permalink
https://escholarship.org/uc/item/6qp46020

Author
Synk, Ryan

Publication Date
2019-08-07

Supplemental Material
https://escholarship.org/uc/item/6qp46020#supplemental

Peer reviewed
ABSTRACT
In this poster, a GPU-accelerated sparse multifrontal solver for structurally symmetric matrices is described. The implementation is tested on the Summit supercomputer against the current version, which is parallelized via MPI and OpenMP on CPUs. The GPU-accelerated implementation achieves significant speedup from the original code.

INTRODUCTION
The problem of solving large sparse linear systems is crucially important in high performance computing and is a bottleneck in many engineering and scientific computation algorithms. The Structured Matrices Package (STRUMPACK) is a C++ library used for solving sparse linear systems.

MULTIFRONTAL METHOD
- Created by Duff & Reid in 1983
- Extremely parallelizable
- Breaks large sparse matrix into smaller, dense matrices
- STRUMPACK currently uses this algorithm

ALGORITHM
Multifrontal Method phases:
1. REORDER
2. FACTOR
3. SOLVE

REORDER

FACTOR
Elimination tree is formed, and individual fronts are factored.

SOLVE
A forward and backward substitution is performed to generate the final solution.

IMPLEMENTATION
In order to accelerate the STRUMPACK multifrontal implementation, the dense linear algebra tasks involved with factorizing fronts were offloaded onto the GPU.

ENGINEERING CHALLENGES
- At bottom of tree, there are lots of small fronts
- Memory transfer and allocation for every small front created huge overhead
- Utilized a pool of managed memory that is reused for each level of tree

KERNELS AND STREAMS
Steps of factoring each front:
1. LU factorization of F11 front
2. Triangular solve on F12 and F21 fronts
3. Schur complement update

For small fronts (size less than a given cutoff), all fronts are factored simultaneously using a custom kernel
For large fronts, a CUDA Stream is used in conjunction with CuBLAS API to hide data transfer overhead:

RESULTS
- Tested using a 3D Poisson Problem of various sizes
- Code tested on Summit — GPU accelerated version ran with 1 IBM Power9 CPU and 1 Tesla V100 GPU
- CPU-parallelized version ran with 7 IBM Power9 CPUs
- Exhibits 2-3 time speedup

ACKNOWLEDGMENTS
Special thanks to my mentor, Pieter Ghysels, for his invaluable help and guidance this summer. Thanks to Donald Wilcox and Kevin Gott for their helpful advice on GPU architectures and CUDA. Thanks to all of the members of the Scalable Solvers Group. This work was prepared in partial fulfillment of the Berkeley Lab Undergraduate Research (BLUR) program, managed by workforce development and education at Berkeley Lab.