A GPU-Accelerated Structurally-Symmetric Sparse Multifrontal Solver

Ryan Synk[1] and Mentor: Pieter Ghysels[1]
Scalable Solvers Group, Lawrence Berkeley Lab

(Dated: 17 March 2020)

In this report, a GPU-accelerated sparse multifrontal solver for structurally symmetric matrices is described. The implementation is created as part of the Structured Matrices Package (STRUMPACK) and makes use of the CUDA API to achieve speedup of the dense linear algebra operations. 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 on a test problem.

I. 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 codes. Linear solvers seek, given a matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, the solution vector $x \in \mathbb{R}^n$ such that $Ax = b$. In many applications, systems where $A \in \mathbb{C}^{n\times n}$, and $x, b \in \mathbb{C}^n$ are also common. A linear system is said to be sparse if most of the entries in the matrix A are zero.

Most algorithms for solving these systems fall under two categories: iterative and direct methods. The work described in this report encompasses the implementation of a direct solver known as the multifrontal method[1]. This algorithm takes the problem of solving a large sparse linear system and converts it to a problem involving dense linear algebra on smaller matrices, known as fronts. This algorithm parallelizes very well, and has since seen many implementations involving parallelism. In 2016, Ghysels et. al proposed a modification to the multifrontal method involving compressing larger fronts via the use of Hierarchically Semi-Seperable (HSS) matrix representations.[2] While HSS matrices are not the subject of the work involved in this project, the software library which came out of the work of Ghysels et. al, called the Structured Matrices Package (STRUMPACK),[3] was the subject of most of the work involved.

In the multifrontal method, much of the computational bottleneck comes from dense linear algebra operations. Parallelism is frequently employed to reduce the overhead of these computations, and this parallelism is usually spread out over various CPU cores via OpenMP or MPI. However, the use of Graphics Processing Units (GPUs) has increased in high-performance computing following the discovery that GPU architectures are well-suited to quickly perform dense linear algebra operations[4,5]. Since then, high-performance computing systems are being built with more and more GPU accelerators in them, and the DOE's newest exascale supercomputer, Frontier, will be built with four times as many GPUs as CPUs[6]. Despite this, the STRUMPACK multifrontal solver currently has no GPU support, and the work performed this summer sought to modify the software package so that it could utilize these resources. Utilizing GPUs to accelerate the factorization of large fronts in the multifrontal method has been the subject of previous study[7,8], and a 2016 paper of Hogg et. al developed an implementation where almost all of the dense linear algebra is performed on the GPU[9]. Most of this previous work has been on symmetric positive-definite and symmetric indefinite cases — these factor $A = LDL^t$ for L unit lower triangular — but the present report focuses on the general structurally-symmetric sparse case, which seeks a factorization $A = LU$ for L unit lower triangular, U upper triangular. Note we take "structurally symmetric" to mean a matrix whose pattern of non-zero entries is symmetric. It was the aim of this work to not just accelerate certain parts of the STRUMPACK multifrontal solver, but to modify the software to put as much work as possible on a GPU.

STRUMPACK is written in C++. In order to take advantage of GPU resources, the code was modified using CUDA, the Nvidia GPU API. Specifically within this API, the code uses CuBLAS, a CUDA API that performs BLAS calls on the device, and CuSolver, which contains kernels that mimic select LAPACK functions. All code was tested using Summit, a supercomputer located in the Oak Ridge National Lab. Each node of Summit contains 6 Nvidia Tesla V100 GPUs, each containing 16GB of memory, a 6MB L2 cache, and a 128KB L1 cache.

The structure of the report is as follows: section II gives an overview of the multifrontal method, section III details changes made to STRUMPACK, section IV gives some numerical results, and section V contains concluding remarks and possible avenues for future study.

II. DETAIL OF ALGORITHM

This section gives a brief outline of the multifrontal method. For a more detailed explanation, see [10], or the original paper of Duff and Reid[1]. Much of the explanation here is borrowed from [2] with the author's permission. The multifrontal method consists of three phases:

- **Reording** - Where the order of the equations in the matrix are changed
- **Factoring** - Which involves an assembly of the frontal matrices and partial factorizations of each front
- **Solving** - Where a forward then backward substitution is performed to obtain a solution

A. Reorder

First, a fill-reducing permutation is applied to A in order to reduce nonzero entries in the L and U factors. Specifically, $A \leftarrow PAP^t$ for a permutation matrix P. This permutation matrix is created by forming the adjacency graph of A, then performing a nested dissection reording. In STRUMPACK, the graph partitioning software METIS[11] is used.

Next, a data structure known as the *elimination tree* is formed. Borrowing the definition from [2]:

Definition II.1 *Assume $A = LU$, where A is an $N \times N$ sparse, structurally symmetric matrix. The elimination tree of A is a tree with N nodes, where the ith node corresponds to the ith column of L and with the parent relations defined by parent$(j) = \min\{i \mid i > j$ and $l_{ij} \neq 0\}$ for $j = 1, \cdots, N-1$*

These nodes are grouped together using the structure of the adjacency graph. Specifically, vertices from the same seperator are grouped in one elimination tree node. These nodes are known as *frontal matrices*, with a 2×2 block structure:

$$F_i = \begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}$$

B. Factor

Once an ordering has been chosen and the tree is formed, the factorization of the matrix begins. This consists of a bottom-up topological traversal of the tree. During this traversal, a node is assembled using information from the nodes in its subtree, and then a partial factorization is applied to the node. The assembly process of a node i in the tree consists of taking the rows and columns of A corresponding to the variables in the F_{11}, F_{12}, and F_{21} blocks and summing them with the extended *update matrices* produced by the children of i:

$$F_i = A_i + \sum_{\nu \in \text{child(i)}} \bar{U_\nu}$$

Where

$$A_i = \begin{pmatrix} A(I_i^{sep}, I_i^{sep}) & A(I_i^{sep}, I_i^{upd}) \\ A(I_i^{upd}, I_i^{sep}) & 0 \end{pmatrix}$$

And $I_i = \{I_i^{sep}, I_i^{upd}\}$ is the index set of row and column indices of F_i corresponding to the global matrix A (after reordering). Once the node has been assembled, a dense partial factorization is performed: $F_{11} = LU$. After this, the next update matrix is formed by taking a Schur complement: $U_i = F_{22} - F_{21}F_{11}^{-1}F_{12}$.

In order to continue traversing the tree, the indices of U_i must be manipulated to match those of its parent in order to create $\bar{U_i}$. As an example, if two children's update matrices, $U_1 = \left(\begin{smallmatrix} a_1 & b_1 \\ c_1 & d_1 \end{smallmatrix}\right)$, and $U_2 = \left(\begin{smallmatrix} a_2 & b_2 \\ c_2 & d_2 \end{smallmatrix}\right)$ have subscript index sets $I_1^{upd} = \{1, 2\}$ and $I_2^{upd} = \{1, 3\}$, then these matrices must be padded with zero rows and columns to match the dimension of the parent:

$$\bar{U_1} + \bar{U_2} = \begin{pmatrix} a_1 & b_1 & 0 \\ a_2 & b_2 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} a_2 & 0 & b_2 \\ 0 & 0 & 0 \\ c_2 & 0 & d_2 \end{pmatrix} = \begin{pmatrix} a_1 + a_2 & b_1 & b_2 \\ a_2 & b_2 & 0 \\ c_2 & 0 & d_2 \end{pmatrix}$$

This operation is known as the "extend-add" operation and can be written as $U_1 \overset{+}{\leftrightarrow} U_2$. This operation can be used to define the relation between frontal matrices and update matrices: for a node i with children $\nu_1, \cdots \nu_q$, we have that $F_i = A_i \overset{+}{\leftrightarrow} U_{\nu_1} \overset{+}{\leftrightarrow} \cdots \overset{+}{\leftrightarrow} U_{\nu_q}$

Once the tree has been traversed and all the factors are generated, the solve phase begins.

C. Solve

This phase finds the solution x to $Ax = b$ by performing a forward substitution using the L factor and a backward substitution using the U factor. The forward solution performs a bottom-up traversal of the tree, while the backward solution uses a top-down traversal.

III. OUTLINE OF SOFTWARE CHANGES

There were a number of changes made to the code of the STRUMPACK library in order to allow for the usage of a GPU. This section details those changes, as well as some of the problems faced in refactoring the code.

A. Data Transfer and Device Memory Overhead

Most of the computational bottleneck in the multifrontal method comes from the dense linear algebra operations performed in the factor phase. In STRUMPACK, this dense linear algebra is performed via BLAS and LAPACK calls. Most of the effort of the project involved finding ways to offload these dense linear algebra operations onto a GPU in order to increase speedup.

This offloading needs to be performed efficiently, however. An initial attempt at utilizing CUDA involved the NVBLAS library, which provides "drop-in" GPU acceleration to a given application. This software package is designed to be linked with an application which makes BLAS calls. It then intercepts those calls and performs them on a GPU automatically. This approach actually made the code much slower. A second attempt at utilizing the CuBLAS API involved allocating/freeing separate device memory and performing separate CuBLAS kernel launches for each front. This, too, greatly increased the factorization time of the code.

The problem here lies in the fact that at the bottom of the elimination tree there can be a massive number of fronts — each of which is usually quite small. Allocating and freeing separate memory for each of these fronts, along with launching the necessary CuBLAS calls created an extraordinarily high overhead for so many small

fronts. The objective became to offload the dense linear algebra to the device while simultaneously minimizing the number of kernel launches, device allocation/free calls, and data transfers between host and device.

In the current model of the code, the factorization proceeds level-by-level in the elimination tree. This is opposed to the recursive elimination tree traversal that is used in the CPU version of STRUMPACK — a level-by-level traversal is more amenable to GPU parallelization. For a given level of the tree, a pool of managed memory is allocated for all of the fronts on that level, and all of the fronts on the level above it. This memory pool is reused throughout the elimination tree, and the memory is only re-allocated if the next level needs more space, which greatly reduces the number of device allocations and frees. The managed memory model allows for automatic CPU/GPU memory transfers. Here, all of the fronts on a given level are loaded onto the GPU, then a kernel is launched which performs the necessary computations. If a given level contains n nodes, then the kernel is launched with n thread blocks, one for each node.

B. Kernel and Streams

Because of the high overhead in performing operations on small fronts, the code behaves differently for larger and smaller fronts. If the fronts on a level are smaller than a set cutoff parameter, then those fronts are all launched simultaneously on the GPU using a custom kernel — each front is assigned a thread block, and within each thread block the factorization is multithreaded. This custom kernel is not nearly as optimized as the NVIDIA-written one, however, so if the fronts on a level are large (larger than a set cutoff parameter), then the fronts are handled using the CuBLAS library.

The custom kernel was designed to perform the necessary dense linear algebra operations for each front. Following the steps of the multifrontal method, this kernel:

1. Performs a partial LU factorization with pivoting on the F_{11} block
2. Triangular solves on the F_{12} and F_{21} blocks
3. Creates Schur complement update

These operations are performed simultaneously for all of the small fronts on a level. Larger fronts are handled via a CUDA stream. Streams allow for serial execution of a series of kernels, and allow for an overlap between computations performed on a GPU and memory transfers from GPU to CPU. For larger fronts, a stream is assigned to each front, and the stream launches the necessary CuBLAS kernels for each step of the factorization of the frontal matrix. Note that there is a limit on how many streams can be launched (and a memory overhead for creating them) which makes them less feasible for the many small fronts towards the bottom of the elimination tree. This necessitates the current split design. Currently, a fixed maximum number of streams is set – for the tests, 20 was used. At each level, these streams are created, and used to process the large fronts. In this way, if the number of large fronts on a level exceeds 20, then those fronts are not processed completely in parallel, but rather factored in chunks of 20.

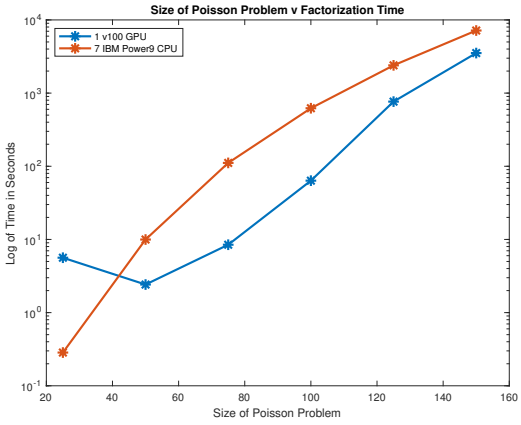

FIG. 1. Factor time results for varying-sized Poisson problems

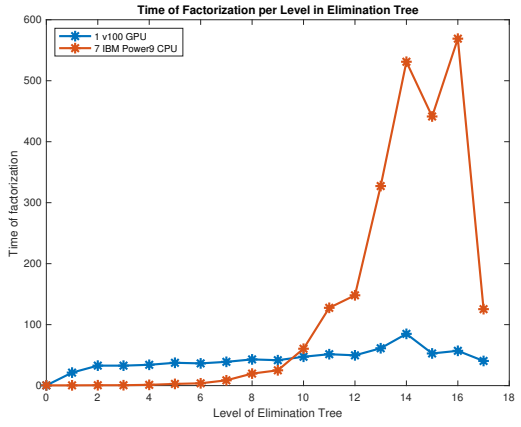

FIG. 2. Factor time per level in the elimination tree

IV. RESULTS

The current state of the code (parallelized only via OpenMP and MPI on CPUs) was tested against the newly-written GPU based code. This code was tested using a 3d Poisson problem with dirichlet boundary conditions. Specifically, we seek a function $u : \mathbb{R}^3 \to \mathbb{R}$ on the unit cube such that $\Delta u = f$ inside the cube and $u = g$ on the boundary of the cube for given functions f and g. The matrix was calculated using the finite difference method on grids of increasing size. In the tests, the CPU-parallelized code was ran on summit using 7 of the IBM Power9 CPUs, while the CUDA code implemented here was ran on Summit using 1 IBM Power9 CPU and 1 Nvidia Tesla V100 GPU.

In figure 1, the number k on the x axis is the size of the mesh used in the finite difference method. In that sense, the size of the linear system that needs to be solved is $k^3 \times k^3$. The y axis is the log of the time taken to factor

the matrix. As the figure shows, the GPU code scales significantly better, being 2-3 times faster for large matrices. That being said, for smaller problems, the CPU code is still faster. This is due to the fact that, when there are many small fronts, transfering the data to and from the GPU is more expensive than actually performing the linear alebra operations. Future work will be performed to optimize the code to perform better on smaller matrices.

Similar results are shown in figure 2. A similarly generated $125^3 \times 125^3$ Poisson system is tested, and the factorization times for various levels of the elimination tree are shown. At the lower levels where there are many small fronts, the CPU version performs slightly better, but eventually is outperformed by the GPU code later in the factorization. Interesting to note is the "dip" at the very end of the graph on the CPU: this is because at the very top of the elimination tree there is only 1 front. On this one front, the multifrontal method only performs an LU factorization, and performs no triangular solve or schur complement dense matrix multiplication. These last two dense linear algebra operations are difficult for a CPU, but the LU factorization can be performed quickly.

V. CONCLUSION

This report described a GPU-accelerated implementation of a sparse multifrontal solver. The GPU-accelerated code, as well as the original version, can be found on the STRUMPACK website[3]. The refactored code achieved higher performance than the original code in a test case involving Poisson's equation.

Further work can be done to improve performance. There are many optimizations that can be added to the custom kernel, and more work needs to be done to improve the speed of the computations on smaller matrices. Additionally, STRUMPACK was designed to incorporate a matrix factorization using Hierarchically Semi-Seperable (HSS) matrix representations, but as it stands right now, none of the HSS factorizations occur in the GPU-accelerated code. Some research in the field has gone into implementing HSS compression algorithms on GPUs, and it could be possible to incorporate these techniques into STRUMPACK as well.

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. To all of the members of the Scalable Solvers Group at the Berkeley Lab, especially Sherry Li, David Brown, Jocelyn Cho, Wissam Sid Lakhdar, and Gustavo Chávez. Lastly, thanks to Workforce Development & Education at the Berkeley Lab, for creating the BLUR program and funding my research.

This work was prepared in partial fulfillment of the requirements of the Berkeley Lab Undergraduate Research (BLUR) Program, managed by Workforce Development & Education at Berkeley Lab.

[1] I. S. Duff and J. K. Reid, ACM Trans. Math. Softw. **9**, 302 (1983).
[2] P. Ghysels, X. Li, F. Rouet, S. Williams, and A. Napov, SIAM Journal on Scientific Computing **38**, S358 (2016), https://doi.org/10.1137/15M1010117.
[3] P. Ghysels, S. Li, G. Chávez, and Y. Liu, "Strumpack," `https://github.com/pghysels/STRUMPACK` (2014).
[4] E. S. Larsen and D. McAllister, in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01 (ACM, New York, NY, USA, 2001) pp. 55–55.
[5] K. Fatahalian, J. Sugerman, and P. Hanrahan, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04 (ACM, New York, NY, USA, 2004) pp. 133–137.
[6] https://www.olcf.ornl.gov/frontier (2019).
[7] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes, in *High Performance Computing for Computational Science – VECPAR 2010*, edited by J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011) pp. 71–82.
[8] G. P. Krawezik and G. Poole (2011).
[9] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, ACM Trans. Math. Softw. **42**, 1:1 (2016).
[10] J. W. H. Liu, SIAM Rev. **34**, 82 (1992).
[11] G. Karypis and V. Kumar, SIAM J. Sci. Comput. **20**, 359 (1998).